RAIDb: Redundant Array of Inexpensive Databases

Emmanuel Cecchet, Julie Marguerite, Willy Zwaenepoel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

RAIDb: Redundant Array of Inexpensive Databases

Emmanuel Cecchet – Julie Marguerite – Willy Zwaenepoel

N° 4921

Septembre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4921--FR+ENG

RAIDb: Redundant Array of Inexpensive Databases

Emmanuel Cecchet[*], Julie Marguerite[†], Willy Zwaenepoel[‡]

Thème 1 – Réseaux et systèmes
Projet Sardes

Rapport de recherche n 4921– Septembre 2003 27 pages

Abstract: Clusters of workstations become more and more popular to power data server applications such as large scale Web sites or e-Commerce applications. There has been much research on scaling the front tiers (web servers and application servers) using clusters, but databases usually remain on large dedicated SMP machines. In this paper, we address database performance scalability and high availability using clusters of commodity hardware. Our approach consists of studying different replication and partitioning strategies to achieve various degree of performance and fault tolerance.
We propose the concept of Redundant Array of Inexpensive Databases (RAIDb). RAIDb is to databases what RAID is to disks. RAIDb aims at providing better performance and fault tolerance than a single database, at low cost, by combining multiple database instances into an array of databases. Like RAID, we define different RAIDb levels that provide various cost/performance/fault tolerance tradeoffs. RAIDb-0 features full partitioning, RAIDb-1 offers full replication and RAIDb-2 introduces an intermediate solution called partial replication, in which the user can define the degree of replication of each database table.
We present a Java implementation of RAIDb called Clustered JDBC or C-JDBC. C-JDBC achieves both database performance scalability and high availability at the middleware level without changing existing applications. We show, using the TPC-W benchmark, that RAIDb-2 can offer better performance scalability (up to 25%) than traditional approaches by allowing fine-grain control on replication. Distributing and restricting the replication of frequently written tables to a small set of backends reduces I/O usage and improves CPU utilization of each cluster node.

Keywords: database, cluster, fault tolerance, performance, scalability, JDBC.

[*] INRIA Rhône-Alpes – Projet Sardes – Emmanuel.Cecchet@inrialpes.fr
[†] ObjectWeb consortium – INRIA Rhône-Alpes – Julie.Marguerite@inrialpes.fr
[‡] EPF Lausanne – IN - Ecublens, CH-1015 Lausanne, Switzerland – willy.zwaenepoel@epfl.ch

Unité de recherche INRIA Rhône-Alpes
655, Avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 – Télécopie : +33 4 76 61 52 52

RAIDb: Redundant Array of Inexpensive Databases

Résumé: Les grappes de machines deviennent de plus en plus utilisées comme plateforme d'exécution pour les applications de type serveur de données comme les sites Web à grande échelle ou les applications de commerce électronique. Beaucoup de travaux de recherche ont été menés pour passer à l'échelle les tiers frontaux (serveurs Web et serveurs d'application) en utilisant des grappes, mais les bases de données restent hébergées sur de grosses machines multiprocesseurs dédiées à cette tâche. Notre approche consiste à étudier différentes stratégies de réplication et de partitionnement pour obtenir différents niveaux de perforrmance et de tolérance aux fautes.

Nous proposons le concept intitulé *Redundant Array of Inexpensive Databases* (RAIDb). RAIDb est aux bases de données ce que RAID est aux disques. RAIDb a pour but de fournir une meilleure performance et tolérance aux fautes qu'une seule base de données, à faible coût, en combinant plusieurs instances de base de données en une matrice de bases de données. Comme RAID, nous définissons plusieurs niveaux de RAIDb qui fournissent différents compromis entre coût, performance et tolérance aux fautes. RAIDb-0 utilise le partitionnement complet, RAIDb-1 offre la réplication complète et RAIDb-2 introduit une solution intermédiaire appelée réplication partielle, dans laquelle l'utilisateur peut définir le degré de réplication de chaque table de la base de données.

Nous présentons une implémentation Java de RAIDb appelée Clustered JDBC ou C-JDBC. C-JDBC fournit à la fois le passage à l'échelle des performances et la haute disponibilité de la base de donnée, au niveau intergiciel, sans changer les applications existantes. Nous montrons, en utilisant le test de performance TPC-W, que RAIDb-2 offre un meilleur passage à l'échelle des performances (jusqu'à 25%) que les approches traditionnelles en permettant un contrôle à grain fin de la réplication. Distribuer et restreindre la réplication des tables accédées fréquemment en écriture à un petit ensemble de machines, réduit les entrées/sorties et améliore l'utilisation du processeur sur chaque nœud de la grappe.

Mots clés: base de données, grappe, tolérance aux fautes, performance, passage à l'échelle, JDBC.

1 Introduction

Nowadays, database scalability and high availability can be achieved, but at very high expense. Existing solutions require large SMP machines or clusters with a Storage Area Network (SAN) and high-end RDBMS (Relational DataBase Management Systems). Both hardware and software licensing cost makes those solutions only available to large businesses.

In this paper, we introduce the concept of Redundant Array of Inexpensive Databases (RAIDb), in analogy to the existing RAID (Redundant Array of Inexpensive Disks) concept, that achieves scalability and high availability of disk subsystems at a low cost. RAID combines multiple inexpensive disk drives into an array of disk drives to obtain performance, capacity and reliability that exceeds that of a single large drive [7]. RAIDb is the counterpart of RAID for databases. RAIDb aims at providing better performance and fault tolerance than a single database, at a low cost, by combining multiple database instances into an array of databases.

RAIDb primarily targets low-cost commodity hardware and software such as clusters of workstations and open source databases. On such platforms, RAIDb will be mostly implemented as a software solution like the C-JDBC middleware prototype we present in this paper. However, like for RAID systems, hardware solutions could be provided to enhance RAIDb performance while still being cost effective.

Clusters of workstations are already an alternative to large parallel machines in scientific computing because of their unbeatable price/performance ratio. Clusters can also be used to provide both scalability and high availability in data server environments. Database replication has been used as a solution to improve availability and performance of distributed databases [2, 12]. Even if many protocols have been designed to provide data consistency and fault tolerance [4], few of them have been used in commercial databases [21]. Gray et al. [9] have pointed out the danger of replication and the scalability limit of this approach. However, database replication is a viable approach if an appropriate replication algorithm is used [1, 14, 26]. Most of these recent works only focus on full database replication. In this paper, we study and compare various data distribution schemes, ranging from partitioning to full replication with an intermediate partial replication solution that offers fine-grain control over replication. We propose a classification in RAIDb levels and evaluate the performance/fault tolerance tradeoff of each solution. The three basic RAIDb levels are: RAIDb-0 for partitioning without redundancy, RAIDb-1 for full mirroring and RAIDb-2 for partial replication. We also explain how to build larger scale multi-level RAIDb configurations by combining the basic RAIDb levels.

We propose C-JDBC, a Java middleware that implements the RAIDb concept. We evaluate the different replication techniques using the TPC-W benchmark [24]. C-JDBC proves that it is possible to achieve performance scalability and fault tolerance at the middleware level using any database engine that has no native replication or distribution support. We show that partial replication offers a significant improvement (up to 25%) compared to full replication by reducing both the communication and the I/O on the backend nodes.

The outline of the rest of this paper is as follows. Section 2 gives an overview of the RAIDb architecture and its components. In section 3, we introduce a classification of the basic RAIDb levels. Then, section 4 shows how to combine those basic RAIDb levels to build larger scale RAIDb configurations. Section 5 presents C-JDBC, a Java implementation of RAIDb and section 6 details the various RAIDb levels implementations. Section 7 describes the experimental platform and an analysis of the benchmark workloads. Experimental results are presented in section 8. Section 9 discusses related work and we conclude in section 10.

2 RAIDb architecture

2.1 Overview

One of the goals of RAIDb is to hide the distribution complexity and provide the database clients with the view of a single database like in a centralized architecture.
Figure 1 gives an overview of the RAIDb architecture. As for RAID, a controller sits in front of the underlying resources. The clients send their requests directly to the RAIDb controller that distributes them among the set of RDBMS backends. The RAIDb controller gives the illusion of a single RDBMS to the clients.

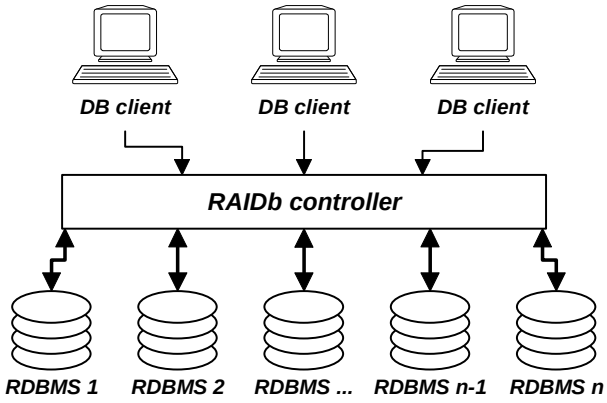

Figure 1. RAIDb architecture overview

2.2 RAIDb controller

RAIDb controllers may provide various degrees of services. The controller must be aware of the database tables available on each RDBMS backend so that the requests can be routed (according to a load balancing algorithm) to the right node(s) by parsing the SQL statement. This knowledge can be configured statically through configuration files or discovered dynamically by requesting the database schema directly from the RDBMS. Load balancing algorithms can range from static affinity-based or round-robin policies to dynamic decisions based on node load or other monitoring-based information.
RAIDb controllers should also provide support for dynamic backend addition and removal which is equivalent to the disks' hot swap feature.
As RAID controllers, RAIDb controllers can offer caching to hold the replies to SQL queries. The controller is responsible for the granularity and the coherence of the cache.
Additional features such as connection pooling can be provided to further enhance performance scalability. There is no restriction to the set of services implemented in the RAIDb controller. Monitoring, debugging, logging or security management services can prove to be useful for certain users.

2.3 Application and database requirements

In general, RAIDb does not impose any modification of the client application or the RDBMS. However, some precautions have to be taken care of, such as the fact that all requests to the databases must be sent through the RAIDb controller. It is not allowed to directly issue requests to a database backend as this might compromise the data synchronization between the backends as well as the RAIDb cache coherency.
As each RDBMS supports a different SQL subset, the application must be aware of the requests supported by the underlying databases. This problem can be easily handled if all RDBMS in-

stances use the same version from the same vendor. For example, a cluster consisting only of MySQL 4.0 databases will behave as a single instance of MySQL 4.0. Nevertheless, heterogeneous databases can be used with RAIDb. A mix of Oracle and PostgreSQL databases is a possible RAIDb backend configuration. In such a case, the application must use an SQL subset that is common to both RDBMS. If the RAIDb controller supports user defined load balancers, the user can implement a load balancer that is aware of the respective capabilities of the underlying RDBMS. Once loaded in the RAIDb controller, the load balancer should be able to direct the queries to the appropriate database.

3 Basic RAIDb levels

We define three basic RAIDb levels varying the degree of partitioning and replication among the databases. RAIDb-0 (database partitioning) and RAIDb-1 (database mirroring) are similar to RAID-0 (disk striping) and RAID-1 (disk mirroring), respectively. Like RAID-5, RAIDb-2 is a tradeoff between RAIDb-0 and RAIDb-1. Actually, RAIDb-2 offers partial replication of the database. We also define RAIDb-1ec and RAIDb-2ec that adds error checking to the basic RAIDb levels 1 and 2, respectively.

3.1 RAIDb-0: full partitioning

RAIDb level 0 is similar to striping provided by RAID-0. It consists in partitioning the database tables among the nodes. Figure 2 illustrates RAIDb-0 with an example of *n* database tables partitioned on 5 nodes. RAIDb-0 uses at least 2 database backends but there is no duplication of information and therefore no fault tolerance guarantees.

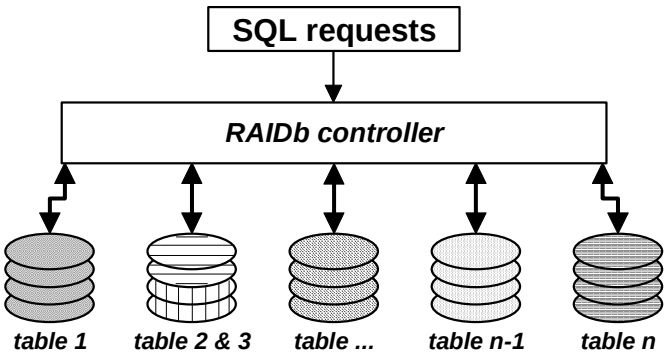

Figure 2. RAIDb-0 overview

RAIDb-0 allows large databases to be distributed, which could be a solution if no node has enough storage capacity to store the whole database. Also, each database engine processes a smaller working set and can possibly have better cache usage, since the requests are always hitting a reduced number of tables. As RAID-0, RAIDb-0 gives the best storage efficiency since no information is duplicated.

RAIDb-0 requires the RAIDb controller to know which tables are available on each node in order to direct the requests to the right node. This knowledge can be configured statically in configuration files or build dynamically by fetching the schema from each database.

Like for RAID systems, the Mean Time Between Failures (MTBF) of the array is equal to the MTBF of an individual database backend, divided by the number of backends in the array. Because of this, the MTBF of a RAIDb-0 system is too low for mission-critical systems.

3.2 RAIDb-1: full replication

RAIDb level 1 is similar to disk mirroring in RAID-1. Databases are fully replicated as shown on figure 3. RAIDb-1 requires each backend node to have enough storage capacity to hold all

database data. RAIDb-1 needs at least 2 database backends, but there is (theoretically) no limit to the number of RDBMS backends.

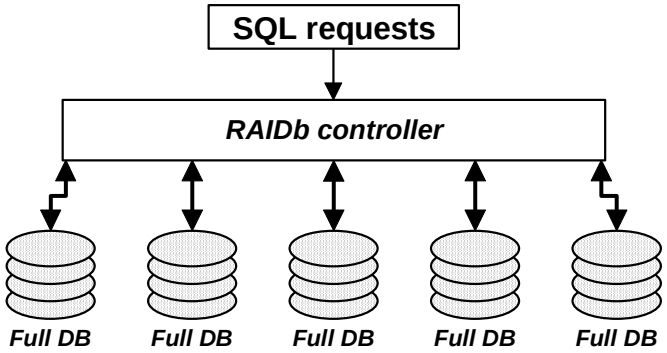

Figure 3. RAIDb-1 overview

The performance scalability will be limited by the capacity of the RAIDb controller to efficiently broadcast the updates to all backends. In case of a large number of backend databases, a hierarchical structure like those discussed in section 4 would give better scalability.
Unlike RAIDb-0, the RAIDb-1 controller does not need to know the database schema, since all nodes are capable of treating any request. However, if the RAIDb controller provides a cache, it will need the database schema to maintain the cache coherence.
RAIDb-1 provides speedup for read queries because they can be balanced over the backends. Write queries are performed in parallel by all nodes, therefore they execute at the same speed as the one of a single node. However, RAIDb-1 provides good fault tolerance, since it can continue to operate with a single backend node.

3.3 RAIDb-1ec

To ensure further data integrity, we define the *RAIDb-1ec* level that adds error checking to RAIDb-1. Error checking aims at detecting Byzantine failures [15] that may occur in highly stressed clusters of PCs [10]. RAIDb-1ec detects and tolerates failures as long as a majority of nodes does not fail. RAIDb-1 requires at least 3 nodes to operate. A read request is always sent to a majority of nodes and the replies are compared. If a consensus is reached, the reply is sent to the client. Else the request is sent to all nodes to reach a quorum. If a quorum cannot be reached, an error is returned to the client.
The RAIDb controller is responsible for choosing a set of nodes for each request. Note that the algorithm can be user defined or tuned if the controller supports it. The number of nodes always ranges from the majority (half of the nodes plus 1) to all nodes. If all nodes are chosen, it results in the most secure configuration but the performance will be the one of the slowest backend. This setting is a tradeoff between performance and data integrity.

3.4 RAIDb-2: partial replication

RAIDb level 2 features partial replication which is an intermediate solution between RAIDb-0 and RAIDb-1. Unlike RAIDb-1, RAIDb-2 does not require any single node to host a full copy of the database. This is essential when the full database is too large to be hosted on a node's disks. Each database table must be replicated at least once to survive a single node failure. RAIDb-2 uses at least 3 database backends (2 nodes would be a RAIDb-1 solution).
Figure 4 gives an example of a RAIDb-2 configuration. The database contains 3 tables *x*, *y* and *z*. The first database backend contains the full database, whereas the other nodes host only one or two tables. There is a total of 3 copies for table *x* and *y*, and 2 copies for table *z*. Whichever node fails, it is still possible to retrieve the data from the surviving nodes.

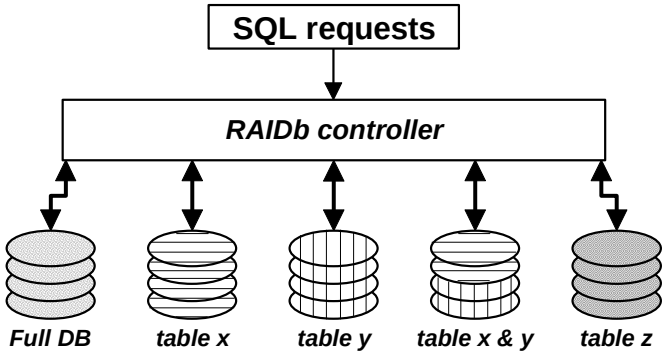

Figure 4. RAIDb-2 overview

Like for RAIDb-0, RAIDb-2 requires the RAIDb controller to be aware of the underlying database schemas to route the request to the appropriate set of nodes.
Typically, RAIDb-2 is used in a configuration where no or few nodes host a full copy of the database and a set of nodes host partitions of the database to offload the full databases. RAIDb-2 can be useful with heterogeneous databases. An existing enterprise database using a commercial RDBMS could be too expensive to fully duplicate both in term of storage and additional licenses cost. Therefore, a RAIDb-2 configuration can add a number of smaller open-source RDBMS hosting smaller partitions of the database to offload the full database and offer better fault tolerance. In figure 4's example, the first backend node on the left could be the commercial RDBMS and the 4 other nodes, smaller open-source databases. These 4 RDBMS can handle a large set of requests and even fail over the large database.
As RAID-5, RAIDb-2 is a good tradeoff between cost, performance and data protection.

3.5 RAIDb-2ec

Like for RAIDb-1ec, RAIDb-2ec adds error checking to RAIDb-2. Three copies of each table are needed in order to achieve a quorum. RAIDb-2ec requires at least 4 RDBMS backends to operate. The choice of the nodes that will perform a read request is more complex than in RAIDb-1ec due to the data partitioning. However, nodes hosting a partition of the database may perform the request faster than nodes hosting the whole database. Therefore RAIDb-2ec might perform better than RAIDb-1ec.

3.6 RAIDb levels performance/fault tolerance summary

Figure 5 gives an overview of the performance/fault tolerance tradeoff offered by each RAIDb level:

- RAIDb-0 offers in the best case the same fault tolerance as a single database. Performance can be improved by partitioning the tables on different nodes, but scalability is limited to the number of tables and the workload distribution among the tables.
- RAIDb-1 gives in the worst case the same fault tolerance as a single database, and performance scales according to the read/write distribution of the workload. On a write-only workload, performance can be lower than for a single node. At the opposite extreme, a read-only workload will scale linearly with the number of backends.
- RAIDb-1ec provides at least the same fault tolerance as RAIDb-1, but performance is lowered by the number of nodes used to check each read query.
- RAIDb-2 offers less fault tolerance than RAIDb-1, but it scales better on write-heavy workloads by limiting the updates broadcast to a smaller set of nodes.
- RAIDb-2ec has better fault tolerance than RAIDb-2 but comes at the price of lower performance and a larger number of nodes.

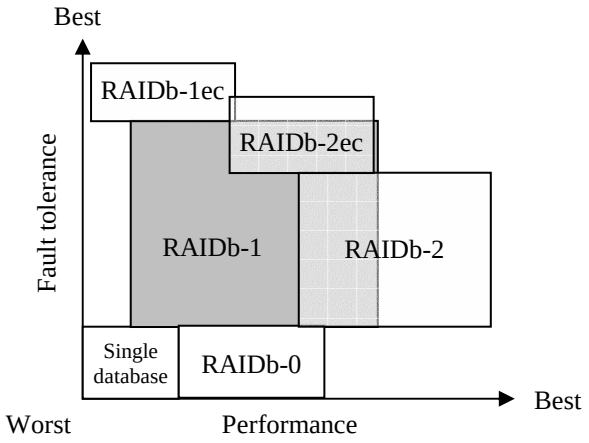

Figure 5. RAIDb performance/fault tolerance tradeoff

4 Composing RAIDb levels

4.1 Vertical scalability

It is possible to compose several RAIDb levels to build large-scale configurations. As a RAIDb controller may scale only to a limited number of backend databases, it is possible to cascade RAIDb controller to support a larger number of RDBMS.

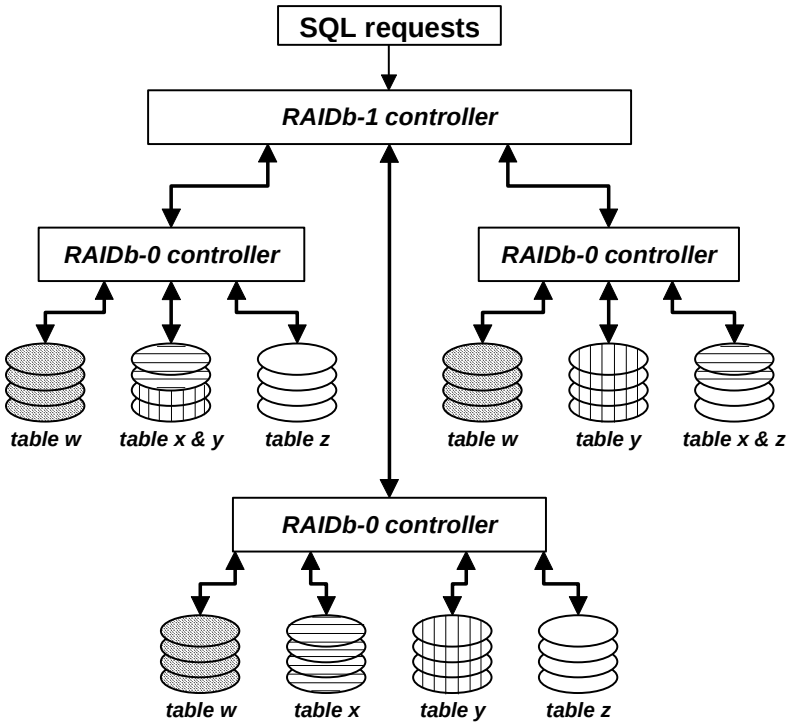

Figure 6. Example of a RAIDb-1-0 composition

Figure 6 shows an example of a 2-level RAIDb composition. The first level RAIDb-1 controller acts as if it had 3 full database backends. At the second level, each full database is implemented

by a RAIDb-0 array with possibly different configurations. Such a composition can be denoted RAIDb-1-0.
Figure 7 gives an example of the same database using a RAIDb-0-1 composition. In this case, the database is partitioned in 3 sets that are replicated using RAIDb-1 controllers. A top level RAIDb-0 controller balances the requests on the 3 underlying RAIDb-1 controllers.

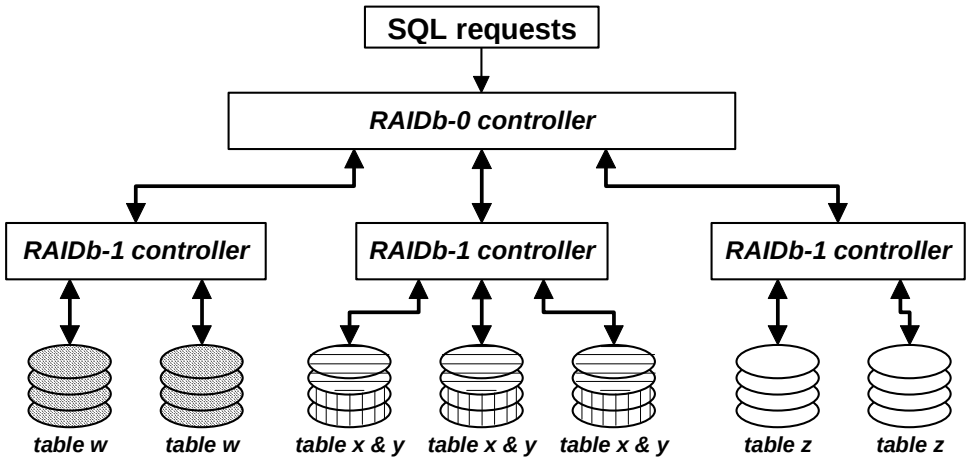

Figure 7. Example of a RAIDb-0-1 composition

There is potentially no limit to the depth of RAIDb compositions. It can also make sense to cascade several RAIDb controllers using the same RAIDb levels. For example, a RAIDb-1-1 solution could be envisioned with a large number of mirrored databases. The tree architecture offered by RAIDb composition offers a more scalable solution for large database clusters especially if the RAIDb controller has no network support to broadcast the writes.
As each RAIDb controller can provide its own cache, a RAIDb composition can help specialize the caches and improve the hit rate.

4.2 Horizontal scalability

The RAIDb controller can quickly become a single point of failure. It is possible to have two or more controllers that synchronize the incoming requests to agree on a common serializable order. Figure 8 gives an overview of the horizontal scalability of RAIDb controllers.

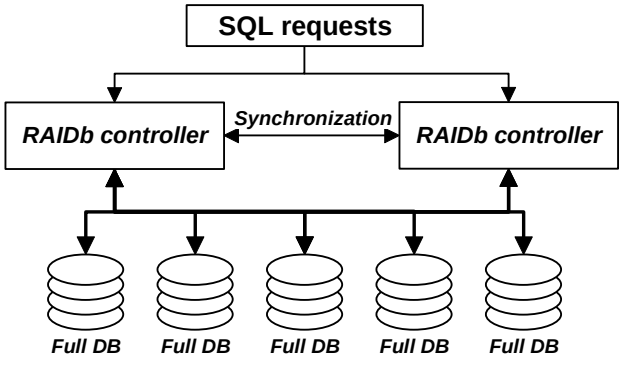

Figure 8. RAIDb horizontal scalability.

Backends do not necessarily have to be shared between controllers as in the above example, but nodes that are attached to a single controller will no longer be accessible if the controller fails.

In the case of shared backends, only one controller sends the request to the backend and notifies the other controllers upon completion.

5 C-JDBC: a RAIDb software implementation

JDBC™, often referenced as Java Database Connectivity, is a Java API for accessing virtually any kind of tabular data [25]. We have implemented C-JDBC (Clustered JDBC), a Java middleware based on JDBC, that allows building all RAIDb configurations described in this paper. C-JDBC works with any existing commercial or open source RDBMS that provides a JDBC driver. The client application does not need to be modified and transparently accesses a database cluster as if it were a centralized database. The RDBMS does not need any modification either, nor does it need to provide distributed database functionalities. The distribution is handled by the C-JDBC controller that implements the logic of a RAIDb controller.

5.1 C-JDBC overview

Figure 9 gives an overview of the different C-JDBC components. The client application uses the generic C-JDBC driver that replaces the database specific JDBC driver. The C-JDBC controller implements a RAIDb controller logic and exposes a single database view, called virtual database, to the driver. A controller can host multiple virtual databases. In the current implementation, the drivers and the controller use sockets to communicate.

The authentication manager establishes the mapping between the login/password provided by the client application and the login/password to be used on each database backend. All security checks can be performed by the authentication manager. It provides a uniform and centralized resource access control.

Each virtual database has its own request manager that defines the request scheduling, caching and load balancing policies. The “real” databases are defined as database backends and are accessed through their native JDBC driver. If the native driver is not capable of connection pooling, a connection manager can be added to perform such a task.

The C-JDBC controller also provides additional services such as monitoring and logging. The controller can be dynamically configured and administered using an administration console that provides XML files describing the controller configuration.

5.2 C-JDBC driver

The C-JDBC driver is a hybrid type 3 and type 4 JDBC driver [25] and it implements the JDBC 2.0 specification. All processing that can be performed locally is implemented inside the C-JDBC driver like in a type 4 JDBC driver. For example, when a SQL statement has been executed on a database backend, the result set is serialized into a C-JDBC driver ResultSet that contains the logic to process the results. Once the ResultSet is sent back to the driver, the client can browse the results locally.

All database dependent calls are forwarded to the C-JDBC controller that issues them on the database native driver like a type 3 JDBC driver. SQL statement executions are the only calls that are completely forwarded to the backend databases. Most of the C-JDBC driver remote calls can be resolved by the C-JDBC controller itself without going to the database backends.

The C-JDBC driver can also transparently fail over multiple C-JDBC controllers implementing horizontal scalability (see section 5.6). The JDBC URL used by the driver is made of a comma separated list of ‘node/port/controller name’ followed by the database name. An example of a C-JDBC JDBC URL is `jdbc:cjdbc://node1:1099:c1,node2:1200:c2/db`. When the driver receives this URL, it randomly picks up a node from the list. This allows all client applications to use the same URL and dynamically distribute their requests on the available controllers.

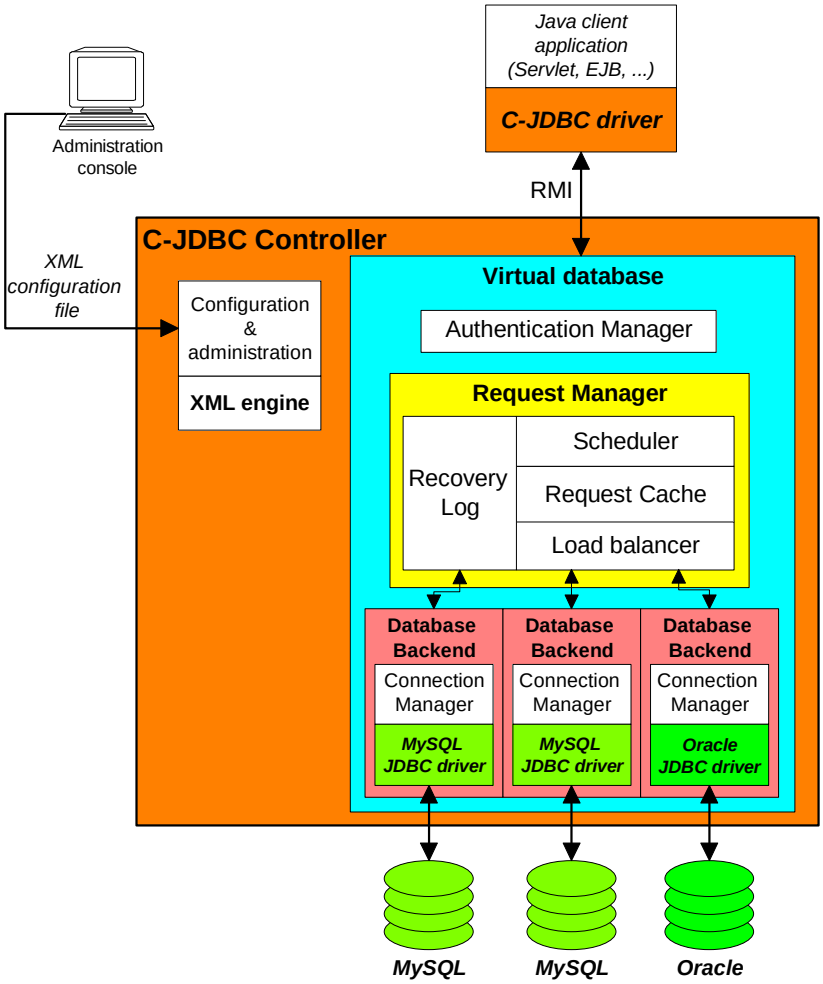

Figure 9. C-JDBC overview

5.3 C-JDBC controller

The C-JDBC controller exports virtual databases to users. A virtual database has a virtual name that matches the database name used in the client application. Virtual login names and passwords match also the ones used by the client. An *authentication manager* establishes the mapping between the virtual login/password and the backend real login/passwords. This allows database backends to have different names and user access rights mapped to the same virtual database.

The request manager is a major component of the C-JDBC controller that implements the RAIDb logic. It is composed of a scheduler, a load balancer and two optional components: a recovery log and a request cache. Each of these components can be superseded by a user-specified implementation.

When a request comes from a C-JDBC driver, it is routed to the request manager associated to the virtual database. The *scheduler* is responsible for ordering the requests according to the desired isolation level. Moreover, consecutive write queries may be aggregated in a batch update so that they perform better. According to the application consistency, some write queries can also be delayed to improve the cache hit rate. Once the request scheduler processing is done, the requests are sequentially ordered.

Then, an optional *request cache* can be used to store the result set associated to each query. We have implemented different cache granularities ranging from table-based invalidations to column-based invalidation with various optimizations. Discussing the query cache design and performance is beyond the scope of this article. Nevertheless, such a cache reduces the request response time as well as the load on the database backends.
If no cache has been loaded or a cache miss occurred, the request finally arrives to the *load balancer*. RAIDb-0 or RAIDb-2 load balancers need to know the database schema of each backend. The schema information is dynamically gathered. When the backend is enabled, the appropriate methods are called on the JDBC DatabaseMetaData information of the backend native driver. Database schemas can also be statically specified by the way of the configuration file. This schema is updated dynamically on each *create* or *drop* SQL statement to reflect each backend schema. Among the backends that can treat the request (all of them in RAIDb-1), one is selected according to the implemented algorithm. Currently implemented algorithms are round robin, weighted round robin and least pending requests first (the request is sent to the node that has the least pending queries).
Once a backend has been selected, the request is sent to its native driver through a connection manager that can perform connection pooling. The ResultSet returned by the native driver is transformed into a serializable ResultSet that is returned to the client by means of the C-JDBC driver.

5.4 Recovery log

C-JDBC implements a recovery log that records all write statements between checkpoints. With each checkpoint corresponds a database dump that is generated by the administrator using the RDBMS specific dump tool. When a backend node is added to the cluster, a dump corresponding to a checkpoint is installed on the node. Then, all write queries since this checkpoint are replayed from the recovery log, and the backend starts accepting client queries as soon as it is synchronized with the other nodes.
The log can be stored on flat files but more interestingly into a database through JDBC. This way, it is possible to have a fault tolerant recovery log by sending the log queries to a C-JDBC controller that will replicate them according to the RAIDb level used. A C-JDBC controller can send the recovery log queries to itself to use the backends to store both database data and recovery log information.

5.5 C-JDBC vertical scalability

It is possible to achieve multiple RAIDb levels by re-injecting the C-JDBC driver into the C-JDBC controller. Figure 10 illustrates an example of a 2 level RAID-x-y composition using C-JDBC.
The top level controller has been configured for RAIDb-x with 3 database backends that are in fact other C-JDBC controllers. The C-JDBC driver is used as the backend native driver to access the underlying controller. Each backend is in fact a RAIDb-y array implemented by other C-JDBC controllers. Therefore, it is possible to build any composition of RAIDb configurations by simply configuring each C-JDBC controller with the components implementing the desired RAIDb level. The different level C-JDBC controllers are interconnected using the C-JDBC driver and the database native drivers are used where the real database backends are connected.

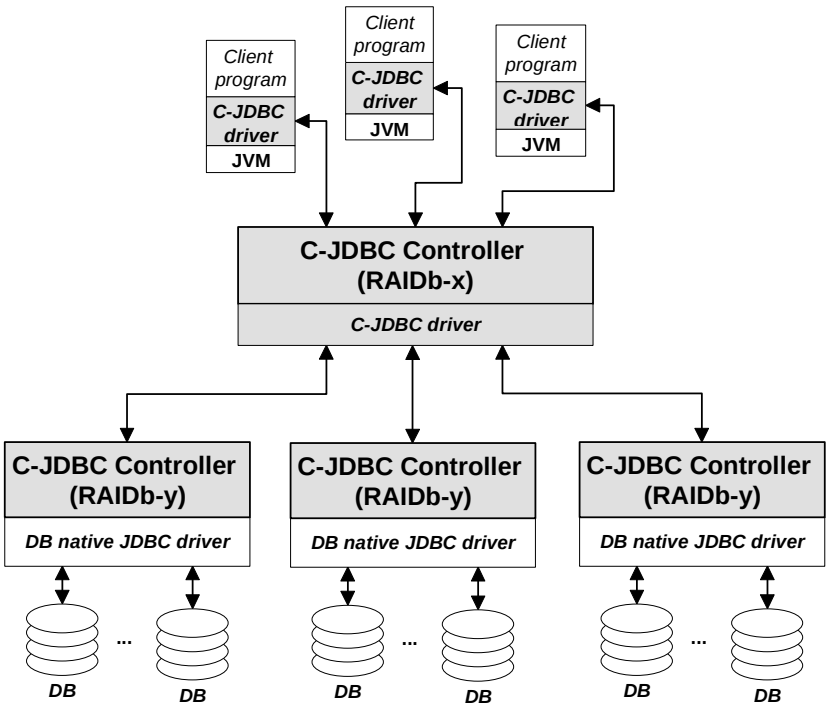

Figure 10. C-JDBC vertical scalability

5.6 C-JDBC horizontal scalability

Horizontal scalability is what is needed to prevent the C-JDBC controller from being a single point of failure. We use the Javagroups [3] group communication library to synchronize the schedulers of the virtual databases that are distributed in several controllers. Figure 11 gives an overview of the C-JDBC controller horizontal scalability.

When a virtual database is loaded in a controller, a group name can be assigned to the virtual database. This group name is used to communicate with other controllers hosting the same virtual database. There is no master/slave mechanism and all backends are truly shared.

C-JDBC relies on Javagroups' reliable and ordered message delivery to synchronize write requests and demarcate transactions. Only the schedulers contain the distribution logic and use group communications. All other C-JDBC components remain the same.

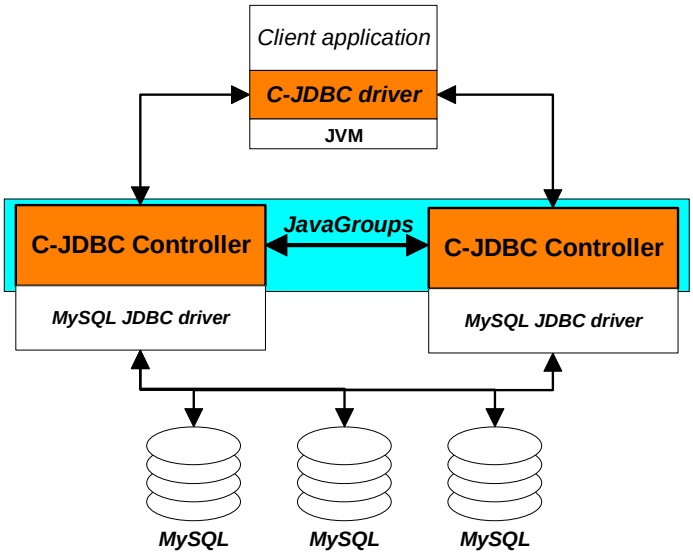

Figure 11. C-JDBC horizontal scalability

6 RAIDb implementations

C-JDBC assumes that the underlying RDBMS provides ACID properties for transactions and that there is always at least one node that has the needed tables to execute a query. For instance, in dynamic content servers, one of the target environments for RAIDb clusters, we know a priori what queries are going to be issued. Therefore, it is always possible to distribute the tables in such a way that all queries can be satisfied (essentially by making sure that for each query there is at least one replica where we have all the tables for that query).

6.1 RAIDb-0

RAIDb-0 requires the request to be parsed to know which tables are needed to execute the request. When the RAIDb-0 load balancer initializes, it fetches the database schema from each backend using the appropriate JDBC calls. When a request needs to be executed, the load balancer checks which backend has the tables needed by the request. As no table is replicated, only one backend can serve each request.

When the client starts a transaction, connections are started lazily on each backend when the first request is sent to this node. If the requests always hit the same backend, only one connection is dedicated for the transaction. In the worst case where a transaction requires data from every backend, a connection per backend is dedicated for the transaction until it commits or rollbacks. If a transaction spans multiple backends, the controller waits for all backends to commit or rollback before notifying the client. A two-phase commit is needed to ensure that all backends commit or rollback.

The user can define the policy to adopt for a ‘CREATE TABLE’ request, either creating the table on a specific node or choosing one node from a set of nodes using a round-robin or random algorithm. The load balancer dynamically updates the schema of each backend on each create/drop statement so that requests always can be forwarded to the appropriate backends.

6.2 RAIDb-1

RAIDb-1 usually does not require parsing the requests, since all backends have a full copy of the database and can therefore execute any query. When a client issues a CREATE TABLE statement, the table is created on every node.

We have implemented optimistic and pessimistic transaction-level schedulers with deadlock detection. To detect the deadlocks, we need to know the database schema and parse the requests to check which tables are accessed by each query. We also have a simple query level scheduler that let the backends resolve the deadlocks.
One thread is dedicated to each backend to send write requests sequentially. The load balancer ensures 1-copy serializability [4] and post write queries in the same order in each thread queue. The user can define if he wants to send the result as soon as one, a majority or all backends complete the request. If one backend fails, but others succeeded to execute the write request, then the failing backend is disabled, because it is no more coherent. The administrator will have to restore the database in a state corresponding to a known checkpoint (using a dump, for example). Then the recovery log will replay all writes restarting from the checkpoint and re-enable the backend once its state is synchronized with the other nodes.
As for the RAIDb-0 load balancer, connections are started lazily. After a write query inside a transaction, one connection per backend has been allocated for the transaction. Transaction commit or rollback use the same principle as write queries. The user can define if he wants only one, a majority or all nodes to commit before returning. If one node fails to commit, but others succeed, the failing node is automatically disabled.
Finally, read requests are executed on a backend according to a user defined algorithm. We have implemented round-robin (RR), weighted round-robin (WRR) and least pending requests first (LPRF) which selects the node with the fewest pending queries (which should be approximately the less loaded node in an homogeneous environment).

6.3 RAIDb-2

Like RAIDb-0, RAIDb-2 needs to maintain a representation of each backend database schema. The query has to be parsed to be routed to the right set of backends.
We have implemented the same set of schedulers as RAIDb-1. Read and write queries are implemented almost the same way as in RAIDb-1 except that the requests can only be executed by the nodes hosting the needed tables. The set of nodes is computed for each request to take care of failed or disabled nodes.
Unlike RAIDb-1, when a node fails to perform a write query or to commit/rollback a transaction, it is not disabled. In fact, only the tables that are no longer coherent are disabled (that is to say, removed from the backend schema). If a commit/rollback fails on one node, all tables written on this node during the transaction are disabled. This way, RAIDb-2 allows continued service by a backend, after a partial failure which leaves most of its tables up-to-date.
Completion policy is also user definable and can be completely synchronous (wait for all nodes to complete) or more relaxed by waiting only for a majority or just the first node to complete.
New database table creation policy can be defined the same way as RAIDb-0, but an additional "set of nodes" parameter is taken into account. In RAIDb-0, the table is only created on one node, but RAIDb-2 allows any degree of replication for a table. The user can define a set of nodes where the table can possibly be created and a policy determining how to choose nodes in this set. The policy can range from all nodes to a fixed number of nodes (at least 2 nodes to ensure fault tolerance) chosen according to a selectable algorithm (currently random and round-robin). Those policies prove to be useful to distribute and limit the replication of created tables in the cluster.

6.4 Current limitations

C-JDBC assumes that all databases are in a coherent state at startup. C-JDBC does not handle the replication of databases to build the initial cluster state. An external ETL (Extraction, Transformation and Loading) tool such as Enhydra Octopus [8] should be used to build the replicated initial state.

RAIDb-2 controllers allow partial failures, but we do not currently support partial recovery for failed tables. The backend must be completely disabled to restore all tables from a known checkpoint using the recovery log.

7 Experimental environment

7.1 TPC-W Benchmark

The TPC-W specification [24] defines a transactional Web benchmark for evaluating e-commerce systems. TPC-W simulates an online bookstore. We use the Java servlets implementation from University of Wisconsin [5] with the patches for MySQL databases. As MySQL does not support sub-selects, each such query is decomposed as follows: the result of the inner select is stored in a temporary table; then, the outer select performs its selection on the temporary table and drops it after completion.

The database manages ten tables: *customers*, *address*, *orders*, *order_line*, *shopping_cart*, *shopping_cart_line*, *credit_info*, *items*, *authors*, and *countries*. The *shopping_cart* and *shopping_cart_line* tables store the contents of each user shopping cart. The *order_line*, *orders* and *credit_info* tables store the details of the orders that have been placed. In particular, *order_line* stores the book ordered, the quantity and discount. *Orders* stores the customer identifier, the date of the order, information about the amount paid, the shipping address and the status. *Credit_info* stores credit card information such as its type, number and expiry date. The *items* and *authors* tables contain information about the books and their authors. Customer information, including real name and user name, contact information (email, address) and password, are maintained in the *customers* and *address* tables.

Of the 14 interactions specified in the TPC-W benchmark specification, six are read-only and eight have update queries that change the database state. The read-only interactions include access to the home page, new products and best-sellers listings, requests for product detail, and two search interactions. Read-write interactions include user registration, updates to the shopping cart, two purchase interactions, two involving order inquiry and display, and two administrative updates.

TPC-W specifies three different workload mixes, differing in the ratio of read-only to read-write interactions. The browsing mix contains 95% read-only interactions, the shopping mix 80%, and the ordering mix 50%. The shopping mix is considered the most representative mix for this benchmark. The database scaling parameters are 10,000 items and 288,000 customers. This corresponds to a database size of 350MB, which fits entirely in the main memory of database server.

7.2 Measurement Methodology

For each workload, we use the appropriate load generator and record all SQL requests sent by the application server to the database. The resulting trace file contains all transactions received by the database during one hour.

We have written a multithreaded trace player that replays the trace as if requests were generated by the application server. This way, we can test every configuration with the exact same set of queries.

We always use a separate machine to generate the workload. The trace player emulates 150 concurrent client sessions. We first measure the throughput of a single database without C-JDBC. Then, we evaluate the various C-JDBC configurations using a dedicated machine to run the C-JDBC controller.

To measure the load on each machine, we use the *sysstat* utility [23] that every second collects CPU, memory, network and disk usage from the Linux kernel. The resulting data files are analyzed post-mortem to minimize system perturbation during the experiments. Specific C-JDBC controller profiling is performed in separate experiments using the OptimizeIt profiler [17].

7.3 Software Environment

The TPC-W benchmark trace is generated using Apache v.1.3.22 as the Web server and Jakarta Tomcat v3.2.4 [11] as the servlet server.
The Java Virtual Machine used for all experiments is IBM JDK 1.3.1 for Linux. We always use a pessimistic transaction level scheduler in C-JDBC controllers. We experiment two different load balancing algorithms: round-robin (RR) and least pending requests first (LPRF). All experiments are performed without query caching in the controller.
We use MySQL v.4.0.8gamma [16] as our database server with the InnoDB transactional tables and the MM-MySQL v2.0.14 type 4 JDBC driver.
All machines run the 2.4.16 Linux kernel.

7.4 Hardware Platform

We use up to six database backends. Each machine has two PII-450 MHz CPU with 512MB RAM, and a 9GB SCSI disk drive[4]. In our evaluation, we are not interested by the absolute performance values but rather by the relative performance of each configuration. Having slower machines allows us to reach the bottlenecks without requiring a large number of client machines to generate the necessary load.
A number of 1.8GHz AMD Athlon machines run the trace player and the C-JDBC controllers. We make sure that the trace player does not become a bottleneck in any experiment. All machines are connected through a switched 100Mbps Ethernet LAN.

7.5 Configurations

7.5.1 SingleDB

This configuration directly uses the MySQL native JDBC driver on a single database backend without using C-JDBC. This reference measurement is referred to as SingleDB in the experimental reports.

7.5.2 RAIDb-0

Figure 12 summarizes the table dependencies resulting from the queries performing a join between multiple tables. Except the *shopping_cart* table, all tables have a relation with each other either directly or indirectly. Therefore, the only available configuration for RAIDb-0 will be to store all tables except *shopping_cart* on one node and *shopping_cart* on a separate node. Without support for distributed joins, RAIDb-0 configurations are restricted to distribution of disjoint tables which could lead, like in the TPC-W example, to a very poor distribution.

[4] These machines could seem old but they have a CPU vs I/O ratio comparable to recent workstations.

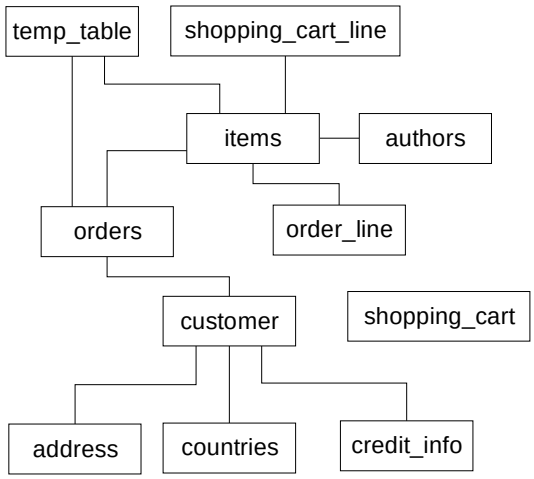

Figure 12. TPC-W table dependencies for joins.

The temporary table created for sub-selects is populated with information from the *orders* table, therefore we adopt a policy where temporary tables are created on the node hosting the *orders* table.

7.5.3 RAIDb-1

There is no choice about data placement with RAIDb-1 since the whole database is replicated on each node. We present results using two different load balancing algorithms for read queries: RAIDb-1 RR uses a simple round-robin distribution whereas RAIDb-1 LPRF uses the least pending request first distribution defined in 6.2. For the write queries, we choose a completion policy that returns the result as soon as one backend has completed the execution of the query.

7.5.4 RAIDb-2

Table 1 details the request distribution on the different tables for each TPC-W mix. If a request spans over multiple tables, it is counted once for each table. Therefore the total percentage can be greater than 100%. Total number of requests shows that for the browsing and shopping mixes, more than 50% of the requests are joins.

It is also interesting to note that the percentage of write queries are correlated but do not correspond to the percentage of write interactions. Actually, the browsing, shopping and ordering mixes have 5, 20 and 50% of write interactions, respectively. But the resulting database write queries represent 13, 20 and 28% of the overall requests, respectively. However, the writes distribution on database tables greatly varies according to the mix.

There is clearly a hotspot on the *items* table for the browsing and shopping mixes. It is also the most accessed table but to a lesser extent for the ordering mix. Even in this last mix, the write threshold is very moderate and as this table is involved in many joins, it will benefit from being replicated on every node.

The *authors* table is the second most accessed table for the browsing and shopping mixes. Both tables are mostly read accessed and therefore would benefit from a large replication on every cluster node. The small read-only *countries* table will also benefit from a full replication. The *address* table is also a read mostly table that can be replicated everywhere.

When shifting to a workload with a higher write ratio, the hits on both *items* and *authors* tables decrease sharply whereas *shopping_cart_line* and *customers* tables get more than 45% of the overall accesses for the ordering mix. *Orders, order_line, credit_info,* and *shopping_cart* are write mostly tables especially for the ordering mix where some of them are nearly write only tables. Replication of these tables has to be limited and distributed among the backends.

Table 2 summarizes the table replication in the different RAIDb-2 configurations ranging from 3 to 6 nodes. Note that the temporary table that is used to implement sub-selects can only be created on the nodes having a copy of the tables. The *customers*, *address*, *items*, *authors* and *countries* tables are replicated on all nodes. These settings apply well for the shopping and ordering mixes, however they are not necessary for the browsing mix.
The heaviest query in term of CPU usage it the best seller query that performs a join between 5 tables (*orders*, *order_line*, *items*, *authors* and the temporary table). This query can only be executed on the nodes having a copy of these tables. The best seller query occurs 4566, 2049 and 261 times for the browsing, shopping and ordering mixes, respectively. Restricting the orders table replication for the browsing mix induces a performance penalty and result in load imbalance. We have measured a performance drop of 42% when only half of the nodes can perform the best seller query. Therefore we choose to replicate all tables needed for the best seller query in the browsing mix only.

Table 1. TPC-W workload: read and write requests distribution on database tables.

Table name	Browsing mix			Shopping mix			Ordering mix		
	total	*read*	*write*	*total*	*read*	*write*	*total*	*read*	*write*
customers	4.5 %	4.1 %	0.4 %	9.0 %	7.5 %	1.4 %	21.3 %	17.5 %	3.9 %
address	1.2 %	1.1 %	0.1 %	2.8 %	2.6 %	0.2 %	7.7 %	7.4 %	0.3 %
orders	8.3 %	0.6 %	7.6 %	5.0 %	1.4 %	3.6 %	6.5 %	3.2 %	3.2 %
order_line	7.7 %	7.4 %	0.4 %	4.2 %	3.2 %	1.0 %	3.3 %	0.3 %	3.1 %
credit_info	0.5 %	0.1 %	0.4 %	1.0 %	0.3 %	0.7 %	3.1 %	0.1 %	3.1 %
items	86.7 %	86.2 %	0.5 %	80.4 %	79.4 %	1.0 %	42.8 %	39.7 %	3.1 %
authors	34.2 %	34.2 %	0 %	25.2 %	25.2 %	0 %	7.9 %	7.9 %	0 %
countries	0.6 %	0.6 %	0 %	1.7 %	1.7 %	0 %	4.1 %	4.1 %	0 %
shopping_cart	2.8 %	0.8 %	2.1 %	9.7 %	1.6 %	8.1 %	5.6 %	0.7 %	4.9 %
shop_cart_line	5.6 %	4.3 %	1.3 %	21.9 %	18.2 %	3.7 %	24.8 %	18.4 %	6.3 %
Total	152.0 %	139.2 %	12.8 %	160.8 %	141.0 %	19.8 %	127.1 %	99.3 %	27.8 %

Table 2. Database table replication for RAIDb-2 configurations.

RAIDb-2 configurations	3 nodes			4 nodes				5 nodes					6 nodes					
	1	*2*	*3*	*1*	*2*	*3*	*4*	*1*	*2*	*3*	*4*	*5*	*1*	*2*	*3*	*4*	*5*	*6*
customers	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
address	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
orders	▨	■	■	▨	▨	■	■	▨	▨	■	■	■	▨	▨	▨	■	■	■
order_line	▨	■	■	▨	▨	■	■	▨	▨	■	■	■	▨	▨	▨	■	■	■
credit_info	▨	■	■	▨	▨	■	■	▨	▨	■	■	■	▨	▨	▨	■	■	■
items	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
authors	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
countries	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
shopping_cart	■		■	■	■			■	■	■			■	■	■			
shop_cart_line	■		■	■	■			■	■	■			■	■	■			
temporary table	▨	■	■	▨	▨	■	■	▨	▨	■	■	■	▨	▨	▨	■	■	■

■ Replicated on this node in all mixes
▨ Replicated in the browsing mix only
Not replicated on this node

Like for RAIDb-1, we present results using two different load balancing algorithms for read queries: RAIDb-2 RR uses a simple round-robin distribution whereas RAIDb-2 LPRF uses the least pending request first distribution defined in 6.2. Table creation policy uses 2 nodes chosen using a round-robin algorithm among the nodes having a copy of *orders* table.

8 Experimental Results

We measure the number of SQL requests performed per minute by each configuration. We only report the best result of three runs at the peak point for each configuration.

8.1 Browsing mix

Figure 13 shows the throughput in requests per minute as a function of the number of nodes for each configuration using the browsing mix. As expected, the RAIDb-0 configuration with 2 nodes peaks at 138 requests per minute, just 9 requests per minute more than the single database configuration that saturates at 129 requests per minute. The lack of distribution opportunities for RAIDb-0 does not allow to get better performance.
RAIDb-1 RR starts with a linear speedup with a throughput of 261 requests per minute with 2 nodes. The 6 nodes configuration reaches 542 requests per minute, representing a speedup of 4.2. RAIDb-1 LPRF achieves 628 requests per minute due to a better load balancing. However, the speedup remains below 5 with 6 nodes. This is due to the implementation of the best seller query. The temporary table needs to be created and dropped by all nodes whereas only one will perform the select on this table. This is a good example of the danger of replication.

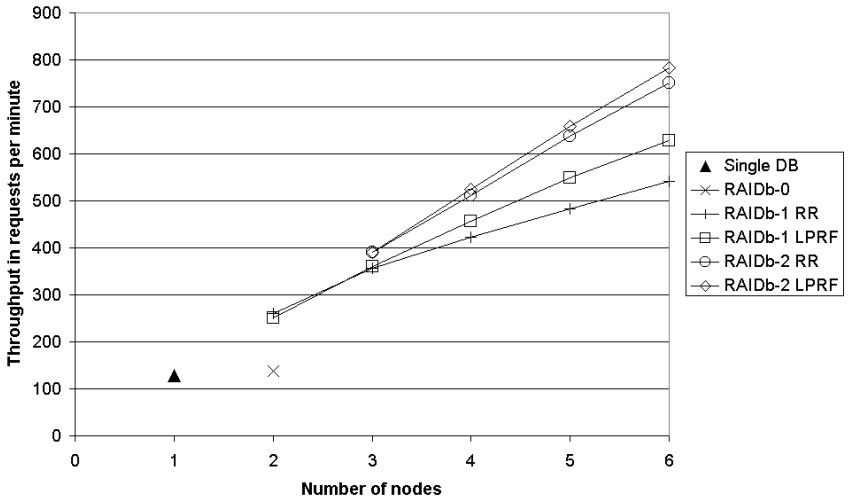

Figure 13. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W browsing mix.

RAIDb-2 configurations limit the temporary table creation to 2 nodes. The results show a better scalability with RAIDb-2 RR achieving 750 requests per minute with 6 nodes (speedup of 5.8). RAIDb-2 LPRF improves RAIDb-1 LPRF performance by 25% and achieves a small superlinear speedup of 6.1 at 784 requests per minute. We attribute this good performance to the better temporary table distribution and the limitation of the replication of the shopping cart related table.

8.2 Shopping mix

Figure 14 reports the throughput in requests per minute as a function of the number of nodes for the shopping mix, which is often considered as the most representative workload. The single database without C-JDBC achieves 235 requests per minute at the peak point.

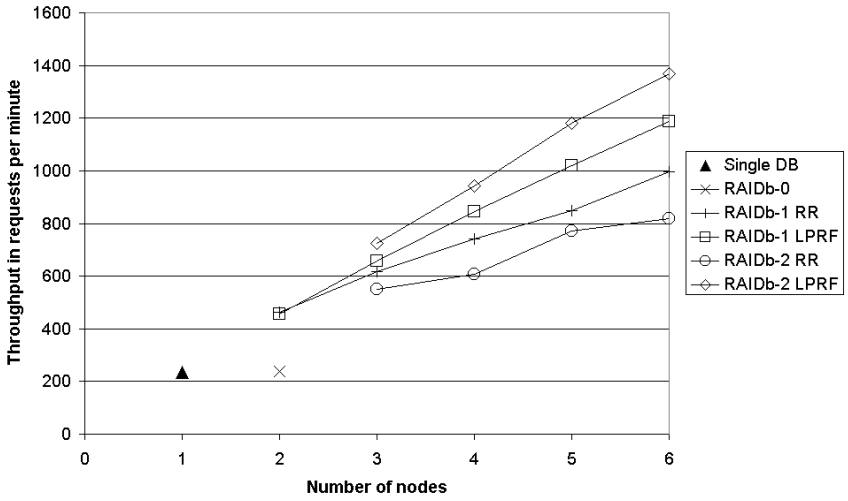

Figure 14. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W shopping mix.

RAIDb-0 suffers from its unbalanced distribution and peaks at 240 requests per minute. RAIDb-1 RR scalability is similar to the one observed for the browsing mix with a peak at 996 requests per minute with 6 nodes. RAIDb-1 LPRF performs better mainly due to the reduction of the best seller queries compared to the browsing mix. RAIDb-1 LPRF achieves 1188 requests per minute with 6 nodes.

RAIDb-2 RR gives the least scalable performance. We will explain the problem using the table replication distribution presented in table 2 with the 6 nodes configuration. When the load balancer wants to execute a query on the *orders*, *order_line* and *credit_info* tables and its current index is positioned on node 1, 2 or 3, the index is moved to the next available node having these tables, namely node 4. The same phenomenon appears with any of the shopping cart tables that will be executed by node 1 if the index is currently on node 4, 5 or 6. We notice a ping-pong effect of the index between nodes 1 and 4.

We can reduce this effect by alternating the table replication order. Instead of replicating shopping cart related tables on nodes 1, 2 and 3 we can replicate them on nodes 1, 3 and 5. Therefore, order related table replicas will be moved from nodes 4, 5 and 6 to nodes 2, 4 and 6. With this new configuration, we obtain a throughput of 887 requests per minute which is better than the previous RAIDb-2 RR configuration saturating at 820 requests per minute. But still, RAIDb-2 with a round robin load balancing algorithm remains the least scalable configuration (among the configurations using replication).

RAIDb-2 LPRF shows the benefits of fine grain partial replication over full replication with 1367 requests per minute at the peak point with 6 nodes. With this dynamic load balancing algorithm, partial replication provides a linear speedup up to 5 nodes. The 6 nodes setup achieves a speedup close to 5.9.

8.3 Ordering mix

Figure 15 shows the results for the ordering mix for each configuration. Almost all queries on the *shopping_cart* table are small writes and their execution on a separate node does not improve performance in the RAIDb-0 configuration.

.

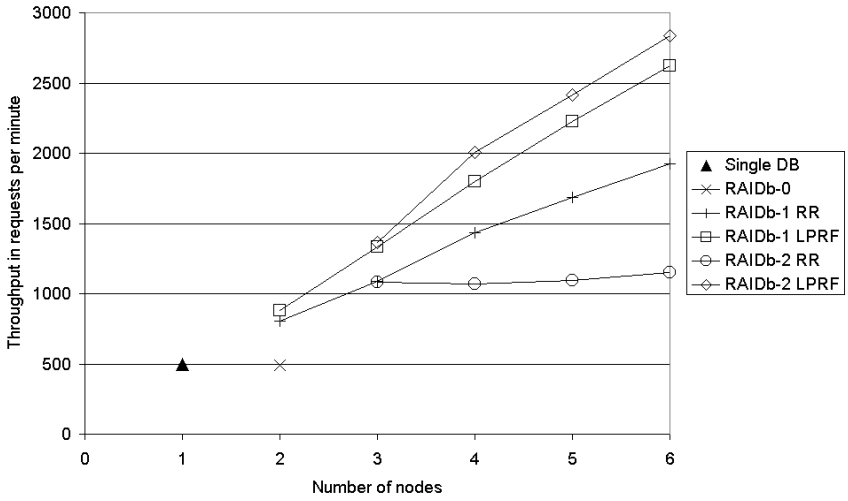

Figure 15. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W ordering mix.

We observe that round robin load balancing gives poor performance for RAIDb-1 and becomes a real bottleneck for RAIDb-2. Even when trying to reduce the ping-pong effect using the alternate distribution used for the shopping mix, we obtain a throughput of 1561 requests per minute with 6 nodes compared to 1152 requests per minute for the original RAIDb-2 RR configuration.
The load imbalance of the round robin algorithm is accentuated when the workload becomes more write intensive. Simple algorithms such as Least Pending Request First alleviate this problem and give significantly better results. The improvement from RAIDb-1 RR to RAIDb-1 LPRF is 700 requests per minute, from 1923 to 2623 requests per minute. RAIDb-2 LPRF achieves 2839 req/min with 6 nodes offering the best throughput of all tested configurations for this mix

8.4 Summary

RAIDb-0 just offers database partitioning and does not provide performance scalability for workloads having hotspots on one table.
RAIDb-1 performs well on read-mostly workloads where load can be easily balanced, however write performance limits scalability when increasing the number of replicas.
RAIDb-2 allows to tune and control the degree of replication of each table. By limiting the write broadcasts to smaller sets of backends, RAIDb-2 shows always better scalability (up to 25%) over full replication when using a dynamic load balancing algorithm such as Least Pending Request First.
Round-robin load balancing provides poor performance scalability even using a cluster composed of homogeneous nodes. When tables are replicated on a small number of nodes, partial replication becomes very sensitive to load balancing. That is why round-robin is not well suited for partial replication and it becomes a bottleneck for workloads with a high write ratio.

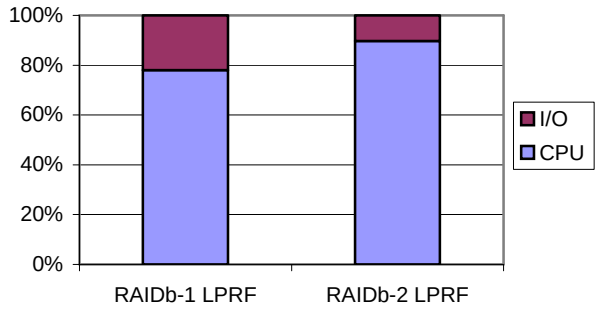

Figure 16. Average CPU vs I/O usage on database backends for RAIDb-1 LPRF and RAIDb-2 LPRF configurations with 6 nodes using the shopping mix.

In all experiments, the average CPU usage on the controller node was below 8% with very little variations between configurations.

If we profile resource usage, we observe that with read-mostly workloads and few number of nodes, the bottleneck is the CPU on the backend nodes. The bottleneck continuously alternates between CPU and disk I/O for workloads involving more writes. Figure 16 shows the average distribution between I/O and CPU usage on the backend nodes for RAIDb-1 LPRF and RAIDb-2 LPRF configurations with 6 nodes for the shopping mix.

The flexibility of partial replication (RAIDb-2) allows reducing the amount of write query broadcasts compared to full replication (RAIDb-1). Therefore, the disk I/O are reduced by an average 11.7% on the backends leaving more time to the CPU to process the requests.

9 Related Work

Since the dangers of replication have been pointed out by Gray et al. [9], several works have investigated lazy replication techniques [19]. The limitations of these approaches are described in [12]. Ongoing efforts on eager replication have also been going on with the recent release of Postgres-R [14]. Several groups are focusing on group communications for asynchronous replication [26] or partial replication [20]. These works are performed at the database level whereas our approach is to implement replication techniques at the middleware level independently of the database engine.

Commercial solutions such as Oracle Real Application Clusters [18] or IBM DB2 Integrated Cluster Environment [**6**] are based on a shared storage system to achieve both performance scalability and fault tolerance. RAIDb targets shared nothing architectures build with commodity hardware.

Existing works in clusters of databases mainly use full database replication. RAIDb also supports partitioning and partial replication. Postgres-R implements basic mechanisms for partial replication [12]. Updates are broadcasted to all nodes that decide whether they have to perform the update or not. RAIDb maintains a knowledge of each backend database schema and broadcast the updates only to the concerned nodes. To the best of our knowledge, our work is the first to evaluate partial replication tradeoffs and to compare its performance with other replication techniques.

Amza et al. have obtained good results with full replication for dynamic content web sites [2]. The approach is similar to the one used in C-JDBC but their implementation is tightly coupled with PHP and MySQL. They do not use transaction markers but require the application programmer to introduce explicit table locks. C-JDBC does not require any application change and can use either transaction markers or explicit locking.

Support for large number of backends usually consists in horizontal scalability where several schedulers synchronize and cooperate [1]. C-JDBC supports both horizontal and vertical scalability allowing different replication policies to be mixed.

10 Conclusion

We have proposed a new concept, called RAIDb (Redundant Array of Inexpensive Databases) that aims at providing better performance and fault tolerance than a single database, at a low cost, by combining multiple database instances into an array of databases. We have defined several levels featuring different replication techniques: RAIDb-0 for partitioning, RAIDb-1 for full replication and RAIDb-2 for partial replication. Additionally, two levels called RAIDb-1ec and RAIDb-2ec provide error checking and tolerate Byzantine failures.
We have presented C-JDBC, a RAIDb software implementation in Java. We have evaluated the performance of the different replication techniques using the TPC-W benchmark on a 6 nodes cluster. RAIDb-0 does not allow the replication of tables that represent a hotspot of the workload. Therefore, performance scalability is very limited. RAIDb-1 scalability achieves a speedup of up to 5.3 with 6 nodes but suffers from the cost of write broadcasts when the number of backends increases. RAIDb-2 allows controlling the degree of replication of each table and prevent database backends from being flooded with writes. RAIDb-2 obtains improvements up to 25% over full replication and achieves linear speedups with read-mostly workloads.
Finally, we have shown that round robin load balancing is not well suited for partial replication especially with write intensive workloads. Simple algorithms such as Least Pending Requests First are sufficient to obtain scalable performance with partial replication.
C-JDBC is an open-source project available for download from http://c-jdbc.objectweb.org/.

11 References

[1] Christiana Amza, Alan L. Cox, Willy Zwaenepoel – Conflict-Aware Scheduling for Dynamic Content Applications – *Proceedings of USITS 2003*, March 2003.

[2] Christiana Amza, Alan L. Cox, Willy Zwaenepoel – Scaling and availability for dynamic content web sites – *Rice University Technical Report TR02-395*, 2002.

[3] Bela Ban – Design and Implementation of a Reliable Group Communication Toolkit for Java – Cornell University, September 1998.

[4] P.A. Bernstein, V. Hadzilacos and N. Goodman – *Concurrency Control and Recovery in Database Systems* – Addison-Wesley, 1987.

[5] Todd Bezenek, Trey Cain, Ross Dickson, Timothy Heil, Milo Martin, Collin McCurdy, Ravi Rajwar, Eric Weglarz, Craig Zilles, and Mikko Lipasti – Characterizing a Java Implementation of TPC-W – *3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads* (CAECW), January 2000.

[6] Boris Bialek and Rav Ahuja – IBM DB2 Integrated Cluster Environment (ICE) for Linux – IBM Blueprint, May 2003.

[7] P. Chen, E. Lee, G. Gibson, R. Katz and D. Patterson – RAID: High-Performance, Reliable Secondary Storage – *ACM Computing Survey*, 1994.

[8] Enhydra Octopus – http://octopus.enhydra.org/.

[9] Jim Gray, Pat Helland, Patrick O'Neil and Dennis Shasha – The Dangers of Replication and a Solution – *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, June 1996.

[10] Monika Henziger – Google: Indexing the Web - A challenge for Supercomputers – *Proceeding of the IEEE International Conference on Cluster Computing*, September 2002.

[11] Jakarta Tomcat Servlet Engine – http://jakarta.apache.org/tomcat/.

[12] Bettina Kemme– Database Replication for Clusters of Workstations – *Ph. D. thesis nr. 13864*, Swiss Federal Institute of Technology Zurich, 2000.

[13] Bettina Kemme and Gustavo Alonso – A new approach to developing and implementing eager database replication protocols – *ACM Transactions on Database Systems*, 2000.

[14] Bettina Kemme and Gustavo Alonso – Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication –*Proceedings of the 26th International Conference on Very Large Databases*, September 2000.

[15] L. Lamport, R. Shostak, and M. Pease – The Byzantine Generals Problem – ACM Transactions of Programming Languages and Systems, Volume 4, Number 3, July 1982.

[16] MySQL Reference Manual – MySQL AB, 2003.

[17] OptimizeIt Profiler – http://www.borland.com/optimizeit/.

[18] Oracle – Oracle9*i* Real Application Clusters – Oracle white paper, February 2002.

[19] E. Pacitti, P. Minet and E. Simon – Fast algorithms for maintaining replica consistency in lazy master replicated databases –*Proceedings of VLDB*, 1999.

[20] A. Sousa, F. Pedone, R. Oliveira, and F. Moura – Partial replication in the Database State Machine – *Proceeding of the IEEE International Symposium on Networking Computing and Applications* (NCA'01), 2001.

[21] D. Stacey – Replication: DB2, Oracle or Sybase – *Database Programming & Design 7*, 12.

[22] I. Stanoi, D. Agrawal and A. El Abbadi – Using broadcast primitives in replicated databases – *Proceedings of ICDCS'98*, May 1998.

[23] Sysstat package – http://freshmeat.net/projects/sysstat/.

[24] Transaction Processing Performance Council – http://www.tpc.org/.

[25] S. White, M. Fisher, R. Cattel, G. Hamilton and M. Hapner – *JDBC API Tutorial and Reference, Second Edition* – Addison-Wesley, ISBN 0-201-43328-1, november 2001.

[26] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme and G. Alonso – Database replication techniques: a three parameter classification – *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, October 2000.